Gerência de Processos em Sistemas Operacionais Distribuídos*

Valdir Rossi Belmonte Filho[1]
Raul Fernando Weber[2]
Antônio Marinho Pilla Barcellos[3]

RESUMO

O módulo de um sistema operacional distribuído encarregado das funções de manipulação de processos é usualmente denominado Gerente de Processos. Suas duas tarefas primordiais, além daquelas encontradas em sistemas centralizados tradicionais, são a manutenção do equilíbrio da carga dos processadores do sistema e a recuperação (até um estado previamente conhecido) de processos situados em um processador que eventualmente tenha falhado, de forma transparente ao usuário. Os algoritmos empregados pelo Gerente de Processos, no entanto, precisam ser cuidadosamente projetados para que não se constituam em escoadouro dos recursos computacionais do sistema. Este trabalho aborda aspectos relacionados à gerência de processos em sistemas operacionais distribuídos tolerantes a falhas, com ênfase em políticas transparentes de recuperação de processos e balanceamento de carga.

ABSTRACT

The module of a distributed operating system in charge of process handling functions is usually called Process Manager. In addition to the functions already found on traditional centralized systems, its two major tasks are the maintenance of processor load and recovery of a process up to a previously known state in the event of a failure, in a transparent manner for the user. However, the algorithms used by the Process Manager must be carefully chosen to keep low the system overhead. Inadequate algorithms may consume a substantial fraction of the total available computing power managing their own internal components, thus reducing system performance. This work deals with aspects related to process management in fault-tolerant distributed operating systems, with emphasis on transparent process recovery and load balancing policies.

Endereço para contato:
Universidade Federal do Rio Grande do Sul
Instituto de Informática
Curso de Pós-Graduação em Ciência da Computação (CPGCC)
Av. Bento Goncalves, 9500 Bloco IV
Caixa Postal 15064 CEP 91501 Porto Alegre, RS
fax: (052) 3365576
tel.: (052) 3368399 (052) 3391355 ramal 6165

---

* Trabalho parcialmente financiado pelo CNPq e IBM Brasil

[1]Bacharel em Ciência da Computação (UFRGS, 1989); Mestrando do CPGCC/UFRGS; Sistemas Operacionais, Processamento Distribuído, Tolerância a Falhas. E-mail: belmonte@inf.ufrgs.br

[2]Engenheiro Elétrico (UFRGS, 1976); Mestre em Computação (UFRGS, 1980); Doutor em Informática, (Karlsruhe, 1986); Tolerância a Falhas, Projeto Automatizado de Sistemas Digitais. E-mail: weber@inf.ufrgs.br

[3]Bacharel em Ciência da Computação (UFRGS, 1990); Mestrando do CPGCC/UFRGS; Sistemas Operacionais , Processamento Distribuído, Arquiteturas Paralelas. E-mail: marinho@inf.ufrgs.br

1 Introdução

Este trabalho define um Gerente de Processos para um sistema operacional distribuído. Além das funções tradicionais existentes em sistemas centralizados, tais como criação e morte de processos locais, a distribuição abre a possibilidade de novas opções, como tolerar falhas de nodos e distribuir a carga entre os componentes do sistema a fim de melhorar o desempenho global do sistema.

Inicialmente, o modelo do sistema é apresentado, seguido pelas políticas e mecanismos de recuperação e balanceamento de carga transparentes empregadas pelo Gerente de Processos. Finalmente, é enfocada a relação e cooperação entre as operações de recuperação e balanceamento.

2 Modelo do Sistema

O modelo do sistema para o qual o Gerente de Processos é proposto baseia-se no desejo de ser o mais genérico possível. Cabe ressaltar, entretanto, que não há um modelo que seja definitivamente superior para todas as situações e necessidades.

Em sistemas de tempo real, há restrições quanto ao tempo máximo decorrido para um processo completar sua execução. Os sistemas são habitualmente estáticos, isto é, o número de processos permanece constante ou varia pouco durante a vida dos mesmos. Para obter eficiência máxima, normalmente as soluções (equipamentos e programas) são amarradas ao problema, dificultando sua generalização. Sistemas orientados a transações, por outro lado, são mais genéricos em razão do menor número de exigências e restrições. Tais sistemas, em sua maioria, empregam equipamentos padronizados, apresentando uma característica dinâmica.

As considerações anteriores conduziram à escolha de um modelo de sistema orientado a transações, de propósito geral. Um outro fator importante para esta escolha reside na pouca disponibilidade de sistemas distribuídos genéricos que suportem tolerância a falhas e balanceamento de carga de forma transparente.

O modelo é não determinista. Os motivos que levam a tal decisão são, além da vantagem óbvia de suportar computação não determinista, a não necessidade de extração de fontes de não determinismo no restante do sistema. Tal extração é tarefa complexa, pela grande variedade de pontos onde há manifestação de não determinismo, de difícil localização [BAB90], como por exemplo o escalonamento de *threads* de um processo.

É importante que o modelo seja capaz de suportar o maior número possível de falhas consecutivas, prosseguindo seu funcionamento de forma degradada. Esta característica é interessante para aumentar a disponibilidade e confiabilidade do sistema.

Finalmente, deseja-se que o Gerente de Processos venha a ser incorporado ao Sistema Operacional Distribuído DIX [BAR90], desenvolvido na UFRGS e que motivou o presente trabalho. Os processos utilizados para comparação de políticas e medição de desempenho ao longo deste artigo foram retirados da versão corrente do DIX, ainda sem o Gerente.

2.1 Arquitetura

O modelo assumido para o sistema conduz a uma arquitetura relativamente simples. Não há necessidade de redundância modular, embora nodos que apresentem tal característica possam ser incluídos no sistema. Assume-se que os processadores são do tipo "falha-parado" (*fail-stop*), isto é, param antes de efetuar uma transição para um estado errôneo. Os nós são fracamente acoplados, ou seja, a comunicação entre os mesmos se dá exclusivamente através de trocas de mensagens, inexistindo compartilhamento de memória. Assume-se que há suporte para memória virtual, preferencialmente através de paginação.

O sub-sistema de comunicação no qual baseia-se o modelo apresenta as seguintes características:

- conta com um protocolo de transmissão que garante o envio e recebimento de todas as mensagens, mantendo a ordem das mesmas (filosofia FIFO);
- provê comunicação 1:N (*multicast* e *broadcast*) eficiente.

→ Primitivas de comunicação 1:N são mais eficientemente implementadas em topologias do tipo barramento ou anel, onde o meio de comunicação é compartilhado por todos os nodos [NI85].

→ Com relação à identificação de processos, assume-se que o núcleo provê mecanismos de identificação transparentes que garantam a continuidade do recebimento de mensagens após a migração, como por exemplo o apresentado em [STE92].

O sistema de arquivos deve dispor de mecanismos de armazenamento estável, a serem empregados para manter informação sobre o estado dos processos (pontos de recuperação - PRs). Esta característica assegura que os pontos de recuperação são salvos em servidores cuja disponibilidade é maior que a dos demais nodos do sistema. A disponibilidade do sistema será determinada, portanto, pela disponibilidade do sistema de arquivos que mantém os pontos de recuperação. O método pelo qual o sistema de arquivos implementa armazenamento estável é transparente ao Gerente de Processos e não será abordado neste artigo. Uma discussão detalhada do assunto é apresentada em [SVO84].

2.2 Detecção de falhas

Considerando-se que os processadores são do tipo falha-parado, é necessário formular um mecanismo para detectar que um nó falhou. Uma das possibilidades é simplesmente não fazer nada, e suspeitar que houve falha quando for tentada comunicação com algum processo do nodo. Entretanto, esta forma possibilita que um nodo fique por um período longo no estado falho sem o mesmo ser detectado, congelando a execução de seus processos durante esse intervalo.

Para possibilitar uma detecção mais rápida, o modelo proposto utiliza um Servidor de *Boot* (SB), semelhante ao empregado no sistema distribuído Amoeba [TAN85]. Quando um nodo é incorporado ao sistema, o mesmo envia uma mensagem ao SB, informando de seu aparecimento. Ambos estipulam uma determinada freqüência com que o nodo enviará mensagens do tipo "Estou vivo" ao SB. Caso o nó demore mais que o estipulado para enviar tal mensagem, o SB tenta comunicação com o mesmo, a fim de certificar-se de que o nodo efetivamente falhou, enviando uma pergunta do tipo "Você está vivo?", com prioridade máxima. Não recebendo resposta dentro de um determinado intervalo, o SB definitivamente declara o nodo como falho. Todos os demais nodos são informados, via *broadcast*, da falha. Assim, os respectivos Gerentes de Processos podem iniciar imediatamente a recuperação de eventuais processos afetados pela falha.

A disponibilidade do SB tem de ser maior que a dos demais nodos. Uma forma de implementá-lo consiste em, por exemplo, selecionar três nodos quaisquer da rede, que cooperam entre si para manter o estado do servidor.

3 Recuperação Transparente

A *recuperação* é a fase na qual o sistema desfaz os eventuais danos causados pela(s) falha(s), colocando o sistema em um estado válido e prosseguindo a execução como se nenhuma falha houvesse ocorrido. Existem duas técnicas básicas para recuperação de erros [CAM86]:

- preditiva (*forward error recovery*, ou recuperação por avanço);
- retroativa (*backward error recovery*, ou recuperação por retorno).

A recuperação preditiva consiste na manipulação do estado atual para atingir um estado livre de erro. O projetista da aplicação, na fase de projeto, define um conjunto de manipuladores (similares a manipuladores de exceção), que descrevem as ações a serem tomadas quando da ocorrência dos erros previstos. Quando determinado erro ocorre, a rotina de tratamento do mesmo é ativada, recolocando o sistema em um estado livre de erro.

Na recuperação retroativa, efetiva-se o retorno do estado de um ou mais processos do sistema a um estado anterior à ocorrência do erro, quando então a computação é reiniciada. Para efetuar o retorno, é necessário dispor dos estados anteriores dos processos. Tais estados são salvos periodicamente, possibilitando sua restauração no futuro.

A recuperação preditiva é totalmente dependente de aplicação, e o seu sucesso depende tanto da capacidade de previsão de todos os possíveis modos de falha quanto da eficácia dos procedimentos de tratamento de erro. Já a recuperação retroativa, contrariamente, é independente de aplicação, suportando qualquer tipo de falha (mesmo as não previstas no projeto), desde que seja detectada. Adicionalmente, o sucesso da recuperação depende exclusivamente da correção do mecanismo de restauração de estado.

O termo *transparência* aqui refere-se ao grau de envolvimento de uma aplicação na obtenção de tolerância a falhas. Um sistema que oferece tolerância de forma transparente não requer nenhuma modificação ou codificação especial nas aplicações. Sistemas tolerantes sem transparência oferecem um conjunto de primitivas que permitem ao projetista efetuar determinados procedimentos de recuperação quando da ocorrência de falhas [CME88].

As vantagens básicas da transparência são a reusabilidade, a portabilidade e a correção [BAC91, BAB90, BOR89, STR85]. O preço a pagar por tais vantagens é uma certa perda de desempenho comparativamente a procedimentos de recuperação específicos para erros bem determinados e conhecidos [CME88]. A tendência, entretanto, é sacrificar essa pequena parcela do desempenho em prol de aplicações mais simples e portáveis, especialmente em sistemas distribuídos orientados a processamento de transações, onde mais importante que o atraso causado pela recuperação é a sua realização bem sucedida [BAB90].

3.1 Seleção da política

Os requisitos de transparência estipulados para o sistema eliminam algoritmos de recuperação preditiva, pelo envolvimento que os mesmos exigem da aplicação na operação de recuperação. Restam algoritmos de recuperação retroativa. Alguns desses algoritmos apresentados na literatura têm suas principais características resumidas na tabela 1.

Tabela 1 - Características dos algoritmos de recuperação retroativa

Método	processador	suporta não determinismo	número de falhas consecutivas	requer armazenamento estável
Mach	falha-parado	não	1	não
Otimista	falha-parado	não	n	sim
Koo	falha-parado	sim	n	sim
MRS	falha-descontrolado	sim	n	sim

Os algoritmos estão ordenados em ordem decrescente de restrições. O sistema Mach [BAB90] apresenta o algoritmo mais restritivo e menos robusto, pois não utiliza armazenamento estável (pontos de recuperação são enviados pela rede para serem armazenados em um nó secundário, que mantém uma cópia *backup* do processo). O sistema não se recupera caso os nós primário e secundário falhem consecutivamente, antes da escolha de um novo secundário.

O algoritmo otimista [STR85] suporta múltiplas falhas consecutivas, porém ainda exige computação determinista, em função de usar histórico de mensagens conjuntamente com ponto de recuperação. O nome do algoritmo deriva da "aposta" que faz com relação à não ocorrência de falhas. Os pontos de recuperação são gravados paralelamente à execução normal, isto é, o sistema não impõe atraso na criação de um PR. Caso uma falha efetivamente ocorra, o processamento terá de retornar ao último PR que efetivamente foi *gravado* em disco (não necessariamente o último *gerado*). A implementação deste algoritmo [JOH91] para o sistema V [CHE88] prevê computação não determinista, sacrificando a transparência. Através de chamadas que a aplicação faz ao sistema, indica-se o tipo de computação (determinista ou não), que define a forma como seu estado é salvo (ponto de recuperação mais histórico de mensagens ou apenas ponto de recuperação, respectivamente).

O algoritmo de Koo [KOO87] incorpora o suporte a não determinismo de forma transparente às aplicações, eliminando o uso do histórico de mensagens. A criação de PRs e a recuperação são empregadas

em algoritmos de duas fases (*two phase commit*). Entretanto, baseia-se em processadores do tipo falha-parado, isto é, assume-se que não ocorre propagação de informação errônea no sistema na ocasião de falhas.

O sistema MRS [RUS80], além de suportar computação não determinista, admite a possibilidade de propagação de erros, permitindo a utilização de processadores do tipo falha-descontrolado. É o método mais genérico.

Na tabela 1, os algoritmos também estão ordenados em ordem decrescente de desempenho. O sistema Mach apresenta o menor tempo de criação de pontos de recuperação, pela não utilização de armazenamento estável. O algoritmo otimista já passa a empregar armazenamento estável, porém o tempo de criação de PRs é diminuído porque o mesmo ocorre assincronamente com o processamento normal. O algoritmo de Koo, ao oferecer suporte para não determinismo, exige uma sincronização maior no instante da criação de pontos de recuperação. O algoritmo MRS, permitindo propagação de erros, tem como conseqüência um grande índice de pontos de recuperação por processo.

A seleção do algoritmo é um compromisso entre generalidade, desempenho e robustez. O sistema Mach apresenta uma baixa robustez, somada à restrição da computação determinista, também exigida pelo algoritmo otimista. Os algoritmos de Koo e MRS suportam não determinismo, apresentando um desempenho um pouco menor. Entre esses dois algoritmos, a diferença resume-se ao tipo de falha assumido para os processadores. O sistema MRS aplicado ao modelo cliente-servidor resulta em praticamente um ponto de recuperação por serviço solicitado (chamadas de sistema). Ou seja, a quantidade de pontos de recuperação é limitada inferiormente pelo padrão de comunicação dos processos componentes do sistema, não dependendo de parâmetros configuráveis ou seletivos em função das necessidades de tolerância a falhas das aplicações. São criados vários pequenos pontos de recuperação, isto é, numa freqüência tal que há poucas páginas modificadas entre dois PRs consecutivos. Tal conduz à desvantagem de a sobrecarga maior ocorrer durante a operação normal (sem falhas) do sistema. Como a falha é a exceção e não a norma, o desempenho final do sistema é menor. O sistema MRS é uma opção interessante quando efetivamente não se dispuser de processadores falha-parado, ou para sistemas cujo padrão de comunicação dos processos não é uniformemente do tipo cliente-servidor, isto é, quando é freqüente a ocorrência da mesma operação em seqüência (vários SEND ou vários RECEIVE consecutivos).

O algoritmo de Koo permite ajustar a freqüência dos pontos de recuperação em função da taxa de falhas do sistema distribuído em questão. A variação desse valor influencia também o tempo de recuperação e o tamanho dos pontos de recuperação, devido à maior distância entre o ponto de falha e os PRs. Não destinado a processamento de tempo-real, o tempo de recuperação pode ser maior (da ordem de segundos) sem que isso prejudique o funcionamento do sistema. A questão do espaço ocupado pelos PRs depende da localidade exibida pelos processos. Alguns processos apresentam uma localidade de execução tão intensa que as páginas modificadas variam muito pouco, mesmo para um longo intervalo de tempo.

O algoritmo de Koo constitui-se, assim, em um ponto intermediário entre generalidade e desempenho. Não é restritivo como o algoritmo usado no sistema Mach e o algoritmo otimista, apresentando um desempenho um pouco inferior, mas suportando computação não determinista. Esse mesmo desempenho, entretanto, é melhor que no algoritmo MRS, apesar da exigência de processadores do tipo falha-parado.

3.2 Política de recuperação

Esta seção descreve a política de recuperação empregada pelo Gerente de Processos, o algoritmo de Koo. Caracteriza-se pela minimização do número de pontos de recuperação por processo, não exigência de processos deterministas e pelo suporte de ocorrência de falhas durante a execução dos algoritmos de recuperação e criação de pontos de recuperação. O algoritmo baseia-se nas seguintes premissas:

- mensagens podem ser perdidas, mas obedecem uma política FIFO com o auxílio de um protocolo de transmissão;
- processadores são do tipo falha-parado;
- após falha de um processo, os demais processos são informados da mesma em um tempo finito. Assume-se que a falha de processos nunca particiona a rede de comunicação.

O método registra no máximo dois pontos de recuperação por processo, resultando em uma economia de armazenamento estável. Há dois tipos de PR: um ponto de recuperação permanente (PRP), definitivamente estabelecido após a execução completa do algoritmo de criação de PR, e um ponto de recuperação tentativo (PRT), que é um PR temporário que existe durante a execução do algoritmo. Quando ocorre algum problema durante a criação de um ponto de recuperação, o PRT é descartado. Caso a criação seja completada, o PRP anterior é descartado e o PRT é promovido a permanente.

Os algoritmos de criação de PR e recuperação são baseados nos protocolos de *commit* em duas fases (*two-phase commit protocols*). No algoritmo de criação, durante a primeira fase, um processo q estabelece PRT e solicita aos demais processos que façam o mesmo. Após receber o resultado de todas as solicitações ("sim" ou "não"), q decide transformar os PRTs em permanentes caso todos os resultados sejam positivos. Caso contrário, q decide descartar todos os PRTs. A ocorrência de falha em um processo corresponde ao envio de um "não" implícito como resposta à solicitação. A segunda fase corresponde à propagação da decisão de q aos demais processos e à execução de tal decisão pelos mesmos. A consistência do sistema é mantida pela garantia de que ou todos os processos criam novos PRPs ou então todos descartam seus respectivos PRTs.

Como um dos objetivos é minimizar o número de processos envolvidos na criação de um ponto de recuperação, o algoritmo é otimizado da seguinte forma: quando um processo p recebe a solicitação de q, aquele somente criará um PRT se o PRT recém-criado por q registrar o recebimento de uma mensagem oriunda de p cujo envio não consta no último PRP de p. A condição anterior evita o surgimento de um estado inconsistente, onde uma mensagem recebida não foi enviada. Note-se que o procedimento é recursivo, isto é, o processo p citado no exemplo anterior também propagará solicitação de criação de um PR tentativo aos demais processos do sistema. Um processo pode identificar a necessidade de criar um PRT através da observação de rótulos que são agregados às mensagens [KOO87]. Para efeito de criação de PRTs, são consideradas apenas as mensagens normais entre processos, isto é, mensagens geradas pelos algoritmos de recuperação e criação não são computadas.

O algoritmo de recuperação baseia-se no mesmo princípio. Na primeira fase, um processo q indaga os demais processos sobre a possibilidade de retornarem aos seus respectivos PRPs. Após receber todas as respostas, q decidirá pelo retorno somente se todas as respostas forem positivas. Na segunda fase, a decisão de q é propagada aos demais processos que a executam. Também aqui a consistência do sistema é garantida, pois todos os processos efetuam a mesma operação. Se a ocorrência de falhas impedir um processo de receber a decisão, este deve bloquear até descobrir qual a decisão do nodo que originou o pedido.

De forma análoga à criação de PRs, o número de processos envolvidos na recuperação também pode ser reduzido. O retorno de um processo q força o retorno de um outro processo p somente se o retorno do primeiro desfizer o envio de uma mensagem sua para p.

Há a possibilidade de um mesmo processo receber solicitações concorrentes, por exemplo, quando executa um pedido de criação de PR e recebe uma solicitação de recuperação, ou vice-versa. Tal fato é denominado *interferência*. Em todas as situações em que há interferência, a solução reside em responder "não" a todas as demais solicitações que chegarem enquanto a primeira não for concluída. Tal medida, entretanto, reduz o grau de concorrência do sistema [KOO87].

A criação de processos pode gerar estados inconsistentes após a recuperação caso não sejam tomadas providências. Considere-se um sistema com dois processos como o da figura 1, onde o processo q é criado por p. Supondo que ambos os processos estejam situados no mesmo nodo, uma falha neste acarretará o retorno de ambos os processos. Se a falha ocorrer no instante t_1, p retornará ao ponto indicado pelo ponto de recuperação PRP1, desfazendo a criação de q. Entretanto, caso a falha ocorra em t_2, após p estabelecer PRP2 (o qual não força q a criar um PR para si), surge uma situação inconsistente: p retorna até PRP2, enquanto q não dispõe de nenhum PR registrado em armazenamento estável. Não há informação suficiente para determinar o novo estado de q.

Uma solução para esse problema é garantir que o sistema nunca retorne aquém do ponto da criação. Considerando-se a criação como uma mensagem do processo pai para o filho, se este imediatamente criar um PR, o pai será forçado a gerar um PR para si (impedindo o surgimento da inconsistência

"criação recebida mas não enviada"). Isto garante que a criação nunca será desfeita, além de evitar inconsistências.

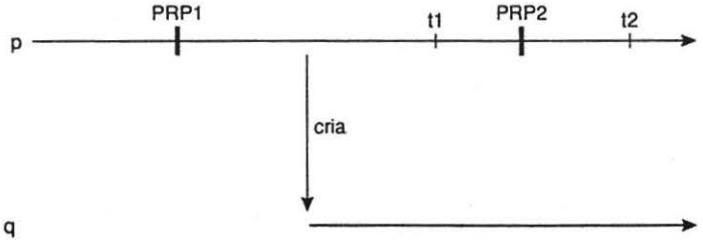

Figura 1 - Inconsistência na criação de processos

A falha de um nó acarreta a parada de todos os processos existentes no mesmo. Quando tal ocorre, é necessário redistribuir esses processos pelos demais nós do sistema, onde então será executado o algoritmo de recuperação para cada um dos processos.

A tarefa da redistribuição está a cargo do Servidor de *Boot*. Ao comprovar que um nó falhou, o SB realiza as seguintes ações: (a) obtém a lista de processos que residiam no nó falho, examinando os pontos de recuperação existentes em armazenamento estável; (b) para cada processo encontrado, solicita ao Gerente de Processos de um outro nó qualquer que passe a recuperá-lo para posterior execução, assumindo a "paternidade" do mesmo. A escolha de qual nó receberá um determinado processo pode ser aleatória ou levar em consideração a política de balanceamento. Para simplificar o algoritmo e diminuir seu tempo de execução total, a escolha aleatória parece suficiente, devido à baixa freqüência da mesma e à possibilidade de rebalancear normalmente o sistema no futuro, durante a execução normal dos processos.

3.3 Mecanismo de recuperação

O Gerente de Processos emprega pontos de recuperação como mecanismo para manter o estado de um processo. Um PR é implementado como sendo o conjunto de páginas modificadas pelo processo desde o último PR (ou desde o início, caso seja o primeiro PR). A identificação de uma página modificada na memória é efetuada através de um bit associado à mesma que a Unidade de Gerência de Memória do processador oferece, também empregado para a política de substituição (*swap*) de páginas.

A política mantém no máximo dois pontos de recuperação por processo. O ponto de recuperação permanente corresponde ao conjunto de todas as páginas modificadas pelo processo até o instante atual. Ao criar um ponto de recuperação tentativo, são registradas as páginas modificadas desde a criação do PRP. Quando o PRT é promovido a permanente, as páginas que o mesmo contém são aplicadas sobre o antigo PRP. Páginas já existentes são substituídas, enquanto que páginas inexistentes são adicionadas ao conjunto. Sobrepondo-se à imagem do processo as páginas que compõe o seu PRP obtém-se o estado do processo no instante da criação do PRP. Além do conjunto de páginas, um PRP contém um campo para auxiliar a recuperação: o identificador do nodo atual do processo, que permite ao SB detectar os processos que executavam no nodo falho e redistribuí-los aos demais nodos.

Considerando-se que o espaço de endereçamento dedicado a código nunca é modificado, o tamanho máximo de um PRP corresponde à totalidade do espaço de endereçamento de dados (estáticos e dinâmicos) de um processo. Supondo-se que um processo modifique todas as suas páginas de dados após criar um PRP, então seu próximo PRT também terá esse mesmo tamanho máximo. Assim, durante o intervalo entre a criação do PRT e a sua promoção a permanente, um processo poderá ocupar no máximo duas vezes o número de páginas de dados de seu espaço de endereçamento. Entretanto, a maioria dos processos exibe forte localidade de execução, resultando em PRs de tamanhos bem menores.

4 Balanceamento de Carga

Balanceamento de carga consiste na distribuição equânime de tarefas entre os nós componentes do sistema distribuído, com a finalidade de obter um ganho de desempenho linear ao número de processadores existentes [LEI91]. Corresponde, igualmente, a uma tentativa de aumentar a produção do sistema, reduzindo seu tempo de resposta médio através da uniformização da carga dos nós [TAN85].

Nesta seção são abordadas a política de balanceamento de carga e seu mecanismo (migração de processos), utilizados pelo Gerente de Processos.

4.1 Seleção da política

O desejo de obter balanceamento transparente elimina políticas de balanceamento estáticas, onde é assumido conhecimento prévio sobre os processos que compõe o sistema distribuído. A política a ser adotada é necessariamente dinâmica, isto é, não há nenhuma informação sobre as características dos processos que serão executados pelo sistema.

Um ambiente formado por redes de estações de trabalho costuma apresentar uma carga heterogênea [THE89]. Há grande número de máquinas ociosas, enquanto outra porção pode estar sobrecarregada. Usualmente, ao invés de processos de longa duração (como simulações), tal ambiente é caracterizado por períodos de ociosidade seguidos da execução de processos curtos que consomem grande quantidade do recurso processador, como por exemplo na seqüência de edição e compilação de um programa.

Entre as políticas dinâmicas encontradas na literatura, duas são localizadas (de Leiss e do Gradiente), enquanto a outra é global (seleção distribuída). Políticas localizadas apresentam a vantagem de seu desempenho ser menos dependente do tamanho total da rede, contrariamente a políticas globais.

O algoritmo de Leiss [LEI91] é localizado, baseando o balanceamento na formação de um *líder*. Sempre que uma vizinhança está desbalanceada, há troca de informação de carga entre os nodos, após o qual é escolhido distribuidamente um nodo, o líder, o qual vai efetivamente realizar uma migração. A formação de um líder apresenta dois inconvenientes: a possibilidade de ocorrência de *deadlocks* e o tempo necessário para concluí-la. *Deadlocks* podem ocorrer em função da forma como um nó é promovido a líder: após receber mensagens de todos os demais vizinhos confirmando sua liderança. Quando dois ou mais nodos entram em estado de recepção de tais mensagens, ocorre o *deadlock*, pois nenhum deles receberá todas as mensagens necessárias para transformar-se em líder. A espera a qual é obrigado o líder também aumenta o tempo total da operação, proporcional ao número de vizinhos.

O algoritmo de Seleção Distribuída [NI85] originalmente é proposto de forma localizada. Sua implementação em um barramento, entretanto, transforma os vizinhos na totalidade da rede. Considere-se um sistema com $L+N+C$ nodos nos estados livre, normal e carregado, respectivamente. Quando um nodo entra no estado carregado, o algoritmo de balanceamento gera um número de mensagens igual a $2L+LC+C+1$ para balancear o sistema. As mensagens estão distribuídas da seguinte forma: (a) nó carregado envia *multicast* ao grupo livre informando seu estado; (b) L mensagens (*multicasts*) dos nodos livres solicitando trabalho; (c) LC respostas dos carregados aos livres; (d) L mensagens dos livres informando aos carregados sobre os processos escolhidos para migração; (e) envio de C processos selecionados para os nodos livres. Portanto, o desempenho do algoritmo, em termos de número de mensagens, é inversamente proporcional ao número de nodos normais. A principal desvantagem do método reside no detalhe de que o número de mensagens é gerado simultaneamente, isto é, criando um pico de utilização do meio de comunicação. O algoritmo é tolerante a falhas, já que a falha de um nodo gera a ocorrência de *time-outs* no recebimento de solicitações/respostas, que usualmente cancelam a participação do nó falho na operação.

O algoritmo do gradiente [LIN87] é localizado, como o de Leiss, porém não exige coordenação entre os integrantes da rede, sem forçar atrasos de sincronização. O desempenho do algoritmo é menos dependente do tamanho total da rede, sendo determinado principalmente pela configuração local dos nodos. Mais especificamente, o número de mensagens (*multicasts*) geradas é diretamente proporcional à distância entre o nodo que inicia uma operação e os nodos livres mais próximos, Considerando-se a implementação do algoritmo em um barramento, os nodos formam um anel lógico (o último nodo do barramento é o vizi-

nho esquerdo do primeiro nodo). Seja d_{esq} a distância (em número de nós) de um nó até o próximo nó livre à esquerda e d_{dir} a distância entre esse mesmo nó e o próximo livre a sua direita. A mudança de estado de um nó gera um número de *multicasts* para estabilizar o sistema igual a:

$$(d_{esq} \text{ div } 2+1)*(d_{dir} \text{ div } 2+1), \text{ de livre para outro qualquer,}$$

$$(d_{esq}+d_{dir}) \text{ div } 2, \text{ de normal ou carregado para livre.}$$

Observa-se que a saída do estado livre domina o custo de comunicação. O algoritmo apresenta uma ordem de grandeza semelhante a do algoritmo de seleção distribuída. A diferença aqui consiste no momento em que as mensagens são geradas. Ao invés de serem enviadas quase simultaneamente, neste algoritmo elas vão sendo geradas pouco a pouco, conforme a informação de proximidade vai atingindo outros nodos. O melhor caso, por outro lado, corresponde a um nó cujos dois vizinhos estão no estado livre, gerando apenas uma mensagem (um *multicast* para os vizinhos, que não propagam a mudança).

A ocorrência de falha em um vizinho afeta o nodo, pois interrompe a propagação das mensagens na direção do nó falho. São possíveis duas alternativas: (a) atribuir proximidade máxima ao vizinho falho, impedindo a migração de processos para esse nó; (b) reorganizar o anel, removendo logicamente o nodo falho.

O algoritmo do gradiente é uma implementação simples de balanceamento localizado, com as vantagens em termos de menor custo de comunicação sem as desvantagens do algoritmo de Leiss. Não apresenta os picos de utilização do meio de comunicação típicos do algoritmo de seleção distribuída, com uma ordem de complexidade aproximada. Tais características conduziram à escolha do algoritmo do gradiente para o Gerente de Processos.

4.2 Política de balanceamento

O algoritmo do gradiente é baseado em uma superfície de pressão, que informa, para cada nodo, a distância (ou *proximidade*) do mesmo até o nodo livre mais próximo. Sempre que um nó muda de estado, sua nova proximidade é propagada aos vizinhos, para que todas as proximidades individuais venham a refletir o novo estado do sistema.

Em uma topologia genérica, um nodo tem vários vizinhos (quatro em uma matriz, por exemplo). Ao entrar no estado carregado, o nodo envia uma solicitação de migração através do vizinho cuja proximidade é menor, isto é, que apresenta o menor caminho até um nodo livre. A solicitação vai sucessivamente se locomovendo de nodo em nodo, até atingir o nodo livre. Quando a topologia empregada é um barramento, surgem algumas peculiaridades. Primeiramente, o número de vizinhos é restrito a dois (esquerdo e direito). Segundo, a proximidade do nodo indica *exatamente* a posição do nodo livre mais próximo. A direção (esquerda ou direita) é a mesma do vizinho de menor proximidade (caso ambos os vizinhos tenham a mesma proximidade, então existem dois nodos livres igualmente próximos do nó em questão). Ou seja, ao invés do nodo carregado enviar uma solicitação ao vizinho, a mesma pode ser entregue diretamente ao nodo livre. Esta peculiaridade auxilia na redução do tráfego de mensagens.

Um barramento apresenta a vantagem do *multicast* eficiente. Um nodo, ao mudar de estado, precisa propagar sua nova proximidade aos vizinhos. Quando a topologia empregada não oferece *multicast*, é necessário enviar uma mensagem individual para cada vizinho. Em um barramento, esta propagação pode ser efetuada com um único *multicast*.

A mensagem de solicitação de migração contém campos que indicam o nó emissor original do pedido e a carga média do processo inicialmente escolhido para ser migrado. Um nó p que recebe uma solicitação de migração oriunda de um nó q e efetivamente está no estado livre aceita o processo e começa tratativas com o emissor para concluir a migração do processo (transferência de estado, etc.). Caso o nó p receba uma solicitação e já tenha saído do estado livre (recebeu processos de outros nodos), a solicitação é reenviada para o nó livre mais próximo de p. Este procedimento perdura até que a solicitação chegue a um nó efetivamente livre ou o sistema fique saturado (inexistam nodos livres). Neste último caso, o emissor original (q) é informado da saturação e conseqüente impossibilidade de completar a solicitação. Note-se que não há necessidade de congelar a execução de processos no nodo carregado. Mesmo sendo escolhido

para uma possível migração, um processo pode continuar sua execução até que a solicitação seja aceita e a migração inicie.

Com o objetivo de aumentar a estabilidade do sistema, optou-se por dar preferência para migração aos processos nascidos no nodo. Assim, um processo que já tenha migrado só poderá migrar novamente se todos os demais processos do nodo também migraram alguma vez. Esta heurística impede que um processo passe mais tempo circulando entre os nodos do sistema do que executando.

4.2.1 Adição e remoção de nodos

Configurando-se o barramento como um anel virtual, então todo o nodo terá dois vizinhos, esquerdo e direito. Cada nodo mantém uma tabela interna com a configuração atual do anel, com pares da forma <identificador do nodo, posição no anel>, o que lhe permite, além da identificação de vizinhos, determinar a identificação dos nodos livres, para enviar solicitações de migração.

A configuração do anel é mantida incrementalmente. Sempre que um nó deseja entrar no sistema, inicialmente deve combinar com o SB protocolos para detecção de falhas (seção 2.2). Após, o novo nodo precisa ser inserido no anel. O SB determina a sua posição física, calculando a nova disposição do anel. Esta disposição do anel é enviada ao novo nó, para que o mesmo possa identificar seus vizinhos e os demais nodos na migração. Por fim, o SB envia um *broadcast* informando que um novo nó entrou no sistema e quem são seus vizinhos (esta informação permite aos demais nodos reprogramarem suas vizinhanças, se afetados pela inserção). Note-se que essa mensagem contém apenas três identificadores: o novo nó e seus dois vizinhos. Uma vez instalado, o novo nodo propaga sua proximidade (igual a zero, pois não possui ainda nenhum processo) aos seus vizinhos, para estabilizar o sistema com a nova configuração.

A remoção de um nodo (devido a uma falha ou outro motivo qualquer) é simples: basta ao SB enviar um *broadcast* informando a remoção de um elemento específico do anel. Como todos os demais nós possuem internamente a representação do anel, os mesmos processam localmente a nova configuração resultante. O ajuste das proximidades é efetuado fazendo-se com que os dois nodos adjacentes ao removido propaguem imediatamente suas proximidades aos novos vizinhos. Este esquema suporta a remoção de vários nodos simultaneamente.

4.2.2 Estimativa de carga

Um nodo em um sistema distribuído pode ter seu tempo de resposta aumentado em função do grau de utilização de dois recursos principais: o processador e a memória. Quanto maior o número de processos, maior o tempo de espera dos mesmos na fila *ready*, enquanto que a escassez de memória gera movimentação de páginas entre a memória principal e a secundária, aumentando o tempo médio de acesso à mesma.

Para medir o consumo do recurso processador, o Gerente de Processos emprega o esquema do "processo *ready*" (carga é indicada pelo intervalo de tempo decorrido entre duas aparições consecutivas do processo na primeira posição da fila "*ready*"), que captura a utilização do processador independente da natureza dos processos, sendo mais precisa do que medidas como número de processos, por exemplo. Para diminuir a influência de picos momentâneos no estado do nó, calcula-se a média das últimas n ativações do processo *ready*. Juntamente a essa medida, utiliza-se o consumo de memória para impedir a ocorrência de *thrashing*. Sempre que a memória disponível for menor que um limite mínimo pré-determinado, o nodo passa para o estado carregado.

4.3 Mecanismo de balanceamento

A migração de processos é o mecanismo que completa a operação de balanceamento. Um dos principais obstáculos da migração é o tempo de transferência do contexto de um processo entre a origem e o destino. Para diminuir esse tempo, é importante evitar a cópia de informação desnecessária, como por exemplo páginas que não serão acessadas pelo processo após a migração. O gráfico da figura 2 mostra uma análise do percentual de páginas acessadas por um processo a partir de diferentes instantes de sua vida, para os processos observados do DIX (isto é, para um dado valor de x, o valor de y representa o percentual de páginas acessadas do instante x até o final da execução).

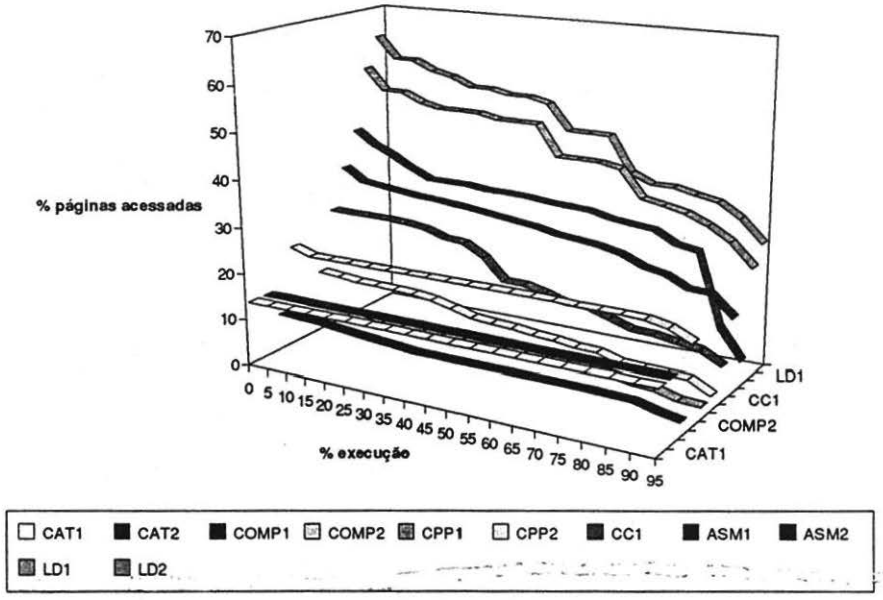

Figura 2 - Padrão de utilização da memória durante a vida de um processo (páginas de 1 Kbyte)

Os processos estudados foram os utilitários `cat`, `compress` e as quatro fases do compilador C: pré-processamento (`cpp`), análise sintático-semântica e geração de código (`cc`), montagem (`asm`) e ligação (`ld`), presentes em sistemas Unix. O sufixo 1 representa a ativação desses processos com um conjunto de dados de entrada mínimo (um programa C "vazio"), enquanto o sufixo 2 refere-se a um conjunto de dados típico (no caso, um programa C de 9 Kbytes). O segundo experimento com o programa `cc` não pode ser exibido em razão de sua análise ter excedido a capacidade do sistema de arquivos da versão do DIX utilizada para tanto.

O gráfico comprova que mecanismos que transferem a totalidade da imagem do processo para o nodo destino são ineficientes, pois ao longo da vida dos processos o percentual de páginas utilizadas pelos mesmos tende a diminuir, em média.

Há dois esquemas que evitam a cópia da totalidade do espaço de endereçamento [ZAY87]: o conjunto residente (CR, que copia todas as páginas do processo que estiverem presentes na memória física do nodo origem) e por demanda (DEM, que não transfere nenhuma página no instante da migração) são mais apropriados. Além da diferença da quantidade de páginas inicialmente transferida, estes dois esquemas diferem em um outro ponto importante: o surgimento de *dependência residual* [DOU91]. No esquema por demanda, o processo passa a depender de ambos os nodos, origem e destino, em razão da existência de páginas modificadas nos mesmos. No esquema do conjunto residente, tal dependência inexiste, pois todas as páginas modificadas são transferidas (juntamente com as demais páginas acessadas que estavam na memória do origem).

A criação do primeiro ponto de recuperação após a migração no esquema DEM precisa tratar do estado do processo em ambos os nodos. O esquema adotado pelo Gerente de Processos é uma alternativa entre o baixo desperdício do esquema DEM e a ausência de dependência residual do esquema CR. Consiste em transferir, no instante da migração, apenas as páginas modificadas, desde o último PR. As demais páginas são transferidas sob demanda, quando solicitadas pelo processo. Quando o origem transfere uma página, a mesma pode ser descartada de seu espaço de endereçamento. O número de páginas transferidas é

menor que no CR, além de não criar dependência residual como o DEM. Esquema semelhante foi empregado no sistema Sprite [DOU91]. Quanto menor o intervalo entre o último PR e a migração, menor o número de páginas transferidas.

Um outro parâmetro a ser determinado é a utilização de busca prévia (*prefetching*) quando ocorre um *page-fault*. Um valor muito pequeno para busca prévia diminui pouco o número total de interrupções, enquanto que um valor muito elevado aumenta o número de páginas transferidas inutilmente. O valor ideal de busca prévia depende do tempo de atendimento de uma interrupção gerada por *page-fault* e da velocidade do meio de comunicação. Conforme examinado nos processos do DIX, o aumento no número de páginas buscadas previamente faz com que o desperdício aumente mais rapidamente que a quantidade de *page-faults* evitados.

5 Recuperação durante Balanceamento

A influência da sobreposição das políticas em um mesmo processo precisa ser analisada. Assume-se que um processo que esteja participando de uma operação de criação de ponto de recuperação ou retorno não está disponível para migração. Similarmente, um processo que está migrando recusa solicitações de criação de PR ou retorno. Entretanto, mesmo com tais premissas é possível que ocorram falhas durante a política e mecanismo de balanceamento, sendo necessário garantir a consistência do sistema após as mesmas.

Há diversas possibilidades de falhas durante a *política* de balanceamento. Uma delas já foi abordada na seção 4.2.1, que corresponde à reorganização do anel quando um nodo falha e pára de enviar sua proximidade. Outra possibilidade é a ocorrência de falha entre o envio da solicitação de migração por um nodo carregado para um nodo livre. É necessário determinar precisamente o instante em que um processo deixa um nodo e passa a outro, para fins de recuperação. Em qualquer instante do protocolo de mensagens utilizado para efetuar a migração, o processo deve pertencer a exatamente um nó (não devem ser permitidos momentos em que o processo não pertença a nenhum nodo ou pertença a ambos). A informação de paternidade de um processo é registrada em armazenamento estável, no PRP (vide seção 3.3), para suportar falhas tanto no nodo origem quanto no destino simultaneamente.

O protocolo baseia-se na ocorrência de *time-outs* para detectar falhas em algum dos lados e cancelar a operação. Até o instante anterior à mudança do PR (*stable_write*) o processo que será migrado ainda pertence ao nodo origem. Desse momento em diante, o processo já trocou logicamente de nodo. Com a operação de escrita o Gerente de Processos do nodo destino registra em armazenamento estável o novo nodo ao qual o processo pertence. Assim, quando o nodo origem envia a mensagem com o estado do processo e não recebe a resposta de confirmação do recebimento do processo são possíveis duas situações: (a) a falha ocorreu *antes* do Gerente do destino registrar o novo local do processo; (b) a falha ocorreu *após* o Gerente do destino registrar o novo local. Observando o valor desse campo, o nodo origem pode determinar se a paternidade do processo mudou. Em (a) o origem continua com o processo, enquanto que em (b) o processo já passou para o destino e a sua recuperação incluirá o processo recém migrado. De forma similar, caso ocorra falha no origem após o envio da mensagem com o estado do processo, o SB vai determinar, examinando o PR, se o processo ainda pertence ao origem no momento da redistribuição dos processos do mesmo. Este procedimento é idêntico para o caso de falha dupla (origem e destino). Isto é, em qualquer instante do protocolo, somente um nodo detém a paternidade do processo.

Uma vez completada a transferência lógica do processo, ainda podem ocorrer falhas durante a transferência física do mesmo (durante a execução do *mecanismo* empregado para tal). O mecanismo de balanceamento escolhido mantém as páginas não modificadas no nodo origem. O processo que migrou não existe mais no origem, restando apenas as páginas do mesmo que ainda não migraram, que estão armazenadas na memória volátil do origem. Caso este venha a falhar, essas páginas serão perdidas após a recuperação. Quando o processo solicitar no destino uma página não transferida, a mesma terá de ser lida do seu ponto de recuperação permanente.

Na hipótese de o nodo destino falhar, o processo migrado é recuperado em um outro nodo, muito provavelmente. Quando sua execução reinicia, o processo pode continuar a acessar as páginas que ainda existam no origem. Acessar uma página na memória de um nodo remoto é uma operação mais rápida do

que lê-la de armazenamento estável. O único detalhe aqui é que há menos páginas no origem, pois todas aquelas transferidas antes da falha foram removidas de sua memória. Uma segunda solicitação dessas páginas terá de ser obrigatoriamente obtida a partir de armazenamento estável.

6 Considerações finais

O Gerente aqui proposto foi simulado em uma rede de estações de trabalho Sun. O protótipo foi desenvolvido em Eiffel, uma linguagem seqüencial orientada a objetos. Tal linguagem foi escolhida em razão da existência de uma biblioteca de rotinas que permite a comunicação entre objetos situados em máquinas distintas [ROS92], fato este que simplificou consideravelmente o desenvolvimento do protótipo. A biblioteca foi desenvolvida na UFRGS, utilizando RPC como primitiva básica de comunicação interprocessos.

O desenvolvimento de programas distribuídos é uma tarefa naturalmente complexa em função da multiplicidade de fluxos de controle e da possibilidade de ocorrência de *deadlocks*. Um possível trabalho futuro consiste na reimplementação do Gerente, desta vez substituindo o paradigma de orientação a objetos pelo de troca de mensagens, aproximando-o cada vez mais de uma versão passível de ser incorporada a um sistema operacional distribuído real.

Bibliografia

[BAB 90] BABAOGLU, Özalp. Fault-tolerant computing based on Mach. **ACM Operating Systems Review**, New York, v.24, n.1, p.27-39, Jan. 1990.

[BAC 91] BACON, David F. Transparent recovery in distributed systems. **ACM Operating Systems Review**, New York, v.25, n.2, p.91-94, Apr. 1991.

[BAR 90] BARCELLOS, Antônio M. P.; BELMONTE FILHO, Valdir R.; LUZ, Marcos V. I.; STEIN, Benhur de O. DIX: **Projeto e implementação de um sistema operacional distribuído para uma rede de estações de trabalho**. Porto Alegre: CPGCC da UFRGS, Out. 1990. (Relatório de Pesquisa).

[BEL 92] BELMONTE FILHO, Valdir R. **Gerência de Processos em Sistemas Distribuídos Tolerantes a Falhas**. Porto Alegre: CPGCC da UFRGS, Fev. 1992. (a ser publicado).

[BOR 89] BORG, Anita; BLAU, Wolfgang; GRAETSCH, Wolfgang; HERRMANN, Ferdinand; OBERLE, Wolfgang. Fault tolerance under UNIX. **ACM Transactions on Computer Systems**, New York, v.7, n.1, p.1-24, Feb. 1989.

[CAM 86] CAMPBELL, Roy H.; RANDELL, Brian. Error recovery in asynchronous systems. **IEEE Transactions on Software Engineering**, New York, v. SE-12, n.8, p.811-826, Aug. 1986.

[CHE 88] CHERITON, David R. The V distributed system. **Communications of the ACM**, New York, v.31, n.3, p.314-333, Mar. 1988.

[CME 88] CMELIK, R. F.; GEHANI, N. H.; ROOME, W. D. Fault tolerant concurrent C: a tool for writing fault tolerant distributed programs. In: SYMPOSIUM ON FAULT TOLERANT COMPUTING SYSTEMS, 18., 1988, Tokyo. **Proceedings...** New York: IEEE Computer Society Press, 1988. 388p.

[DOU 91] DOUGLIS, Fred; OUSTERHOUT, John. Transparent process migration: design alternatives and the Sprite implementation. **Software Practice and Experience**, New York, v.21, n.8, p.757-785, Aug. 1991.

[JOH 91] JOHNSON, David; ZWAENEPOEL, Willy. Transparent optimistic rollback recovery. **ACM Operating Systems Review**, New York, v.25, n.2, p.99-102, Apr. 1991.

[KOO 87] KOO, Richard; TOUEG, Sam. Checkpointing and rollback-recovery for distributed systems. **IEEE Transactions on Software Engineering**, New York, v. SE-13, n.1, p.23-31, Jan. 1987.

[LEI 91] LEISS, E. L.; REDDY, H. N. **Distributed load balancing algorithms: design and performance analysis.** Houston: Research Computation Laboratory, University of Houston, 1991. (Research Report).

[LIN 87] LIN, Frank C. H.; KELLER, Robert M. The gradient model load balancing method. **IEEE Transactions on Software Engineering**, New York, v. SE-13, n.1, p.32-38, Jan. 1987.

[NI 85] NI, Lionel M.; XU, Chong-wei; GENDREAU, Thomas B. A distributed drafting algorithm for load balancing. **IEEE Transactions on Software Engineering**, New York, v. SE-11, n.10, p.1153-1161, Oct. 1985.

[ROS 92] ROSA, Fernando R. **Programação de aplicações distribuídas baseadas em objetos.** Porto Alegre: CPGCC da UFRGS, Nov. 1992. (a ser publicado).

[RUS 80] RUSSEL, David L. State restoration in systems of communicating processes. **IEEE Transactions on Software Engineering**, New York, v. SE-6, n.2, p.183-194, Mar. 1980.

[STE 92] STEIN, Benhur de O. **Projeto do núcleo de um sistema operacional distribuído**. Porto Alegre: CPGCC da UFRGS, Out. 1992.

[STR 85] STROM, Robert E.; YEMINI, Shaula. Optimistic recovery in distributed systems. **ACM Transactions on Computer Systems**, New York, v.3, n.3, p.204-226, Aug. 1985.

[SVO 84] SVOBODOVA, Liba. File servers for network-based distributed systems. **ACM Computing Surveys**, New York, v.16, n.4, p.353-398, Dec. 1984.

[TAN 85] TANNENBAUM, Andrew S.; RENESSE, Robbert van. Distributed Operating Systems. **ACM Computing Surveys**, New York, v.17, n.4, p.419-470, Dec. 1985.

[THE 89] THEIMER, Marvin M; LANTZ, Keith A. Finding idle machines in a workstation-based distributed system. **IEEE Transactions on Software Engineering**, New York, v.15, n.11, p.1444-1458, Sept. 1989.

[ZAY 87] ZAYAS, Edward R. Attacking the process migration bottleneck. **ACM Operating Systems Review**, New York, v.21, n.5, p.13-24, July 1987.